Функции

Часть III
Декораторы

Функции – это объекты

Функции Python относятся к объектам. Их можно **присваивать переменным,** хранить в структурах данных (коллекциях), **передавать их в качестве аргументов** другим функциям и даже **возвращать** их в качестве значений из других функций.

Почему функция является в языке Python объектом ?

Пример присвоения переменной

```
# Поскольку функция render является объектом. Ее можно присвоить еще одной переменной, точно также как это происходит с любым другим объектом.

def render(text):
    print(text.upper() + '!')

render('Hello')
#Hello!

show = render
show('Wellcome')
#Wellcome!

print(show is render)           # True
del render
print(show.__name__)
#render
```

Передача функций в качестве аргумента

Поскольку функции являются объектами, их можно передавать в качестве аргументов другим функциям.

```
def render(text):
    return text.upper() + '!'


def call_hello(func):    ← фукнция более выс. пор.
    result = func('Hello')
    print(result)


>>> call_hello(render)
```

Функция более высокого порядка

Функции которые принимают в качестве аргументов другие функции называют функциями более высокого порядка.

Классический пример таких функций это встроенная функция **map**, которая принимает в качестве аргументов: объект функцию и итерируемый объект. А затем вызывает эту функцию с каждым элементом итерируемого объекта, выдавая результат по мере прохождения итерируемого объекта.

```
def up_cap(text):
    return text.capitalize()


lst = list(map(up_cap, ['cat', 'dog', 'cow']))
print(lst)
```

Функции могут быть вложенными

Python допускает определение функций внутри других функций. Такие функции называются вложенным функциями (nested function) или внутренними функциями(inner function)

```
def speak(text):
    def whisper(t):
        return t.lower() +  '...'
    return  whisper(text)
>>>print(speak('Hello, World'))
```

Всякий раз, когда вы вызываете функцию speak, она определяет новую функцию whisper и затем после этого ее вызывает

Вложенная функция

Внимание! Вложенная функция whisper не существует за пределами функции speak

```
def speak(text):
    def whisper(t):
        return t.lower() +  '...'
    return  whisper(text)


# Попытка вызвать функцию whisper
>>> whisper('Hello, World')
NameError: name 'whisper' is not defined
```

Вопрос! Как получить доступ к вложенной функции whisper за пределами функции speak?

Возврат функции в качестве значения

Не забывайте функции являются объектами – и вы можете вернуть вложенную функцию в качестве значения.

```
def speak():
    def whisper(t):
        return t.lower() +  '...'
    return  whisper
# Получаем из функции speak объект функции whisper
wr = speak()
# Вызываем функцию с одним аргументом
print(wr('Hello, World'))
```

Функции могут захватывать локальные состояния

Перепишем функцию speek следующим образом

```
def speak(text):
    def whisper():
          print(text.lower() +  '...')
    return  whisper

foo = speak('Hello World')
bar = speak('Wellcome')
foo()  → hello, world...
bar()  → wellcome…
```

Внутренние функции получают доступ к родительскому параметру text, определенному в родительской функции. Такой доступ называется лексическим замыканием или для краткости замыканием. Замыкание помнит значения из своего лексического контекста, даже когда поток управления программы больше не находится в этом контексте.

Ключевые выводы

- В Python абсолютно все является объектом, включая функции. Их можно присваивать переменным, передавать в функции более высокого порядка а также возращать из них.
- Функции могуть быть вложенными , и они могут захватывать и уносить с собой часть состояния родительской функции. Функции которые это делают, называются замыканиями.

Python

Function Decorator

Декаратор функции.

Alisa: *Декораторы – они что украсят нашу функцию ?*

Bob: *Нет , декоратор обертывает другую функцию и позволяет исполнять программный код до и после того, как обернутая функция выполниться.*

Декаратор функции.

Alisa: *А что нужно чтобы написать декоратор ?*

Bob: *Объяви функцию декоратор , в нее передай функцию которую будешь обертывать. Реализуй в декораторе вложенную функцию - обертку (wrapper) в котором будет содержатся логика до или после выполнения передаваемой функции. Вызови во вложенной функции , функцию из параметра. Верни вложенную функцию из декоратора.*

Декоратор

```
def decorator_render(func):  ← функция декоратор
    print("Call decorator function")
    def wrapper(text):   ← вложенная функция
        print(f"Логика перед вызовом функции")
        func(text)
        print("Логика после вызова функции")
    return wrapper


def render(text):
    print(f"Это функция render выводит text.upper()}")

# Вызываем декоратор и передаем туда функцию

>>> wrap_func = decorator_render(render)

#Получаем из декоратора вложенную функ. и вызываем её

>>> wrap_func('аргумент1')
```

Аргументы для вложенной функции

```
def decorator_render(func):
    print("Call decorator function")
    def wrapper(*args, **kwargs):
        print(f"Логика перед вызовом функции")
        func(*args, **kwargs)
        print("Логика после вызова функции")
    return wrapper


def render(text):
    print(f"Это функция render выводит text.upper()}")


wrap_func = decorator_render(render)
wrap_func('аргумент1')
```

Оператор @

Вызов декоратора можно переписать так:

```
def decorator_render(func):
    print("Call decorator function")
    def wrapper(text):
        print(f"Логика перед вызовом функции")
        res = func(text)
        print("Логика после вызова функции")
        return res
    return wrapper


@decorator_render

def render(text):
    return f"Это функция render выводит text.upper()}"

render('аргумент1')
```

Для чего использовать декораторы?

- Ведение протокола операции (журналирование)
- Обеспечение контроля за доступом и аутентификацией
- Функции хронометража
- Ограничение частоты вызова API
- Кеширование и др.